A New Verification Method For Embedded Systems

Robert A. Thacker, Chris J. Myers and Kevin Jones
University of Utah
{*thacker,myers,kjones*}*@vlsigroup.ece.utah.edu*

Scott R. Little
Freescale Semiconductor, Inc.
Scott.Little@freescale.com

***Abstract*— Verification of embedded systems is complicated by the fact that they are composed of digital hardware, analog sensors and actuators, and low level software. In order to verify the interaction of these heterogeneous components, it would be beneficial to have a single modeling formalism that is capable of representing all of these components. To address this need, this paper describes an *extended labeled hybrid Petri net* (LHPN) model that includes constructs for Boolean, discrete, and continuous variables as well as constructs to model timing. This paper also presents a method to verify these extended LHPNs. Finally, this paper presents a case study to illustrate the application of this model to the verification of a fault-tolerant temperature sensor.**

I. Introduction

Embedded systems are an unavoidable part of life. In the past, their software has generally been small and frequently written in assembly language. Even though embedded software is now often written in C or other high level languages, such software usually includes embedded assembly code. The effects of this low-level code need to be taken into account. Often things that seem atomic at the higher level become distinct and introduce risky behavior once compiled into assembly. Compilers often do not appropriately treat the low level constructs critical to the proper behavior of these systems [7]. Embedded systems also interact with external analog sensors and actuators, so continuous environment variables must also be considered.

Due to the heterogeneous nature of embedded systems, traditional software testing is often insufficient. *Formal verification*, the process of mathematically analyzing systems to determine their properties, has been shown to be a promising method for validating software [6]. Efforts have been focused in two areas: *static analysis* and *model checking*. Static analysis parses the program to determine its properties structurally. Model checking, on the other hand, creates a representative model and systematically explores all reachable states of the system. These states are then analyzed to determine if invalid reachable states exist. The model constructed is often *abstracted*, removing portions of the system whose complexity does not affect the desired property. The model is also often *decomposed* into simpler subsystems that can be analyzed completely in isolation. An abstract version of these subsystems is then used to analyze the overall system.

In order to apply model checking to embedded systems, it is necessary to develop a single model that is capable of representing both discrete software and continuous interface behavior. *Timed automata* are one candidate, but they require all continuous variables to progress at the same rate, and they do not allow a variable's progress to be stopped. *Hybrid automata* are more expressive, but their use of invariants to ensure progress is a difficult compilation target, as it is not a natural way in which such systems are expressed in higher level languages such as VHDL-AMS and Verilog-AMS. *Hybrid Petri nets* are also considered, but their use of separate continuous places and transitions is again a difficult compilation target from high level languages. Recently, the *labeled hybrid Petri net* (LHPN) model has been developed and applied to the verification of analog and mixed-signal circuits [10], [12], [14]. Compilers have been developed from VHDL-AMS as well as SPICE simulation data [10], [11]. This model includes both Boolean variables for representing digital circuits and continuous variables for representing analog circuits. This paper presents an extended LHPN model that includes discrete variables for representing embedded software variables as well as expressions to check and modify them. These extensions allow for both embedded hardware and software to be represented in a single model. This paper also describes how this new model can be applied to the verification of embedded systems. Finally, this paper presents an algorithm for state space reachability analysis which enables model checking of these extended LHPNs.

This paper is organized as follows. First, Section II describes a motivating example of a fault-tolerant temperature sensor for a nuclear reactor which includes both continuous and discrete variables. Next, Section III introduces the extended LHPN model while Section IV presents its semantics. Since the state space of extended LHPNs is infinite, Section V presents *state sets* which can potentially yield a finite representation of the state space. Section VI describes a reachability method that enables verification of extended LHPN models. Finally, Section VII presents verification results for our motivating example while Section VIII presents our conclusions and future plans.

II. Motivating Example

A traditional hybrid systems example is the cooling system for a nuclear reactor [9], [1]. In this example, the temperature of the nuclear reactor core is monitored, and when the temperature is too high, one of two control rods is inserted to cool the reactor core. After a control rod is used, it must be removed for a set period of time before it can be used again. If the temperature is too high and no control rod is available, the reactor is shut down. In our modified version of the example, there are two temperature sensors to add fault tolerance. Namely, each temperature sensor is periodically sampled and if at any point the temperature difference between them is too large, it is assumed that one

978-1-4244-5028-2/09/$25.00 ©2009 IEEE

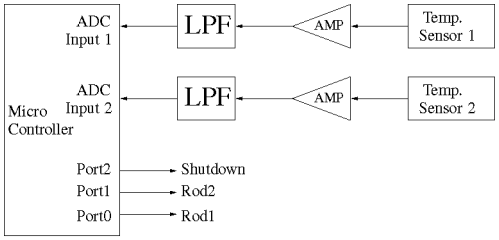

Fig. 1. Fault tolerant cooling system for a nuclear reactor.

of the temperature sensors has become faulty and the reactor is shut down. A block diagram for this fault tolerant cooling system for a nuclear reactor is shown in Fig. 1.

This example is interesting because it includes analog components (i.e., the temperature sensors), mixed-signal components (i.e., the analog/digital converters (ADCs)), digital components (i.e., the microcontroller), and embedded software (i.e., the program running on the microcontroller).[1] The verification problem for this example is to determine if the reactor can be shut down even when the temperature sensors are operating correctly. On the surface, this does not appear to be a problem. However, there are a number of implementation details that make this not so obvious. First, there is typically only one ADC on a microcontroller which is multiplexed to sample from each ADC input one at a time. This means that the temperature sensors are not sampled at exactly the same time. A second problem is that since the comparison of the results is not done with a single atomic instruction at the assembly level, it is possible that the results are not even from the same sampling cycle.

Fig. 2 illustrates an LHPN which models this system. It includes elements to model the environment, the ADC, and the assembly language program. The following sections explain the details and semantics of this model and how it can be analyzed using our new verification method

III. An Extended LHPN Model

An LHPN is a Petri net model originally developed to represent *analog/mixed-signal* (AMS) circuits [10], [12], [14]. This model is inspired by features found in both hybrid Petri nets [4] and hybrid automata [2]. Methods have been developed for generating LHPNs from both a subset of VHDL-AMS [10] and SPICE simulation data [11]. Model checking algorithms have been developed for LHPNs using both explicit *zone-based* methods [10], [12] as well as implicit BDD and SMT-based methods [13]. This paper extends LHPNs to accurately model assembly language level embedded software. Namely, discrete integer values are added to represent register and memory values. An extended expression syntax for enabling conditions and assignments is also introduced to facilitate the manipulations of variables in the model. An extended LHPN is a tuple $N = \langle P, T, B, X, V, F, L, M_0, S_0, Y_0, Q_0, R_0 \rangle$:

- P : is a finite set of places;
- T : is a finite set of transitions;
- B : is a finite set of Boolean variables;
- X : is a finite set of discrete integer variables;
- V : is a finite set of continuous variables;
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;
- L : is a tuple of labels defined below;
- $M_0 \subseteq P$ is the set of initially marked places;
- $S_0 : B \rightarrow \{0, 1, \perp\}$ is the initial value of each Boolean variable;
- $Y_0 : X \rightarrow \{\mathbb{Z} \cup -\infty\} \times \{\mathbb{Z} \cup \infty\}$ is the initial range of values for each discrete variable;
- $Q_0 : V \rightarrow \{\mathbb{Q} \cup -\infty\} \times \{\mathbb{Q} \cup \infty\}$ is the initial range of values for each continuous variable;
- $R_0 : V \rightarrow \{\mathbb{Q} \cup -\infty\} \times \{\mathbb{Q} \cup \infty\}$ is the initial range of rates of change for each continuous variable.[2]

Consider the LHPN for the reactor example shown in Fig. 2. The places are the circles labeled p_0, ..., p_6. The places p_0, p_2, and p_4 are initially marked indicated by the token within the place. The transitions are the boxes labeled t_0, ..., t_7. The flow relation, F, is represented in the figure by the arcs connecting the places and the transitions. This example has one Boolean variable, $shutdown$, which is initially **false**. This example has four discrete variables, A, B, $ADC1$, and $ADC2$ which are all initially undefined (i.e., $[-\infty, \infty]$). Finally, this example has one continuous variable, $temp$, which has an initial value of 2200 and an initial rate of change of -2.

A connected set of places and transitions, or sub-graph, within an LHPN is referred to as a *process*. The LHPN shown in Fig. 2 includes three processes. The process on the left models the temperature of the reactor, the process in the middle models the ADC hardware, and the process on the right models the embedded software.

Before defining the labels formally, let us first introduce the grammar used by these labels. First, the numerical portion of the grammar is defined as follows:

$$\begin{aligned}\chi \quad ::= \quad & c_i \mid x_i \mid v_i \mid (\chi) \mid -\chi \mid \chi + \chi \mid \chi - \chi \mid \chi * \chi \mid \\ & \chi / \chi \mid \chi \hat{} \chi \mid \chi \% \chi \mid \text{NOT}(\chi) \mid \text{OR}(\chi, \chi) \mid \\ & \text{AND}(\chi, \chi) \mid \text{XOR}(\chi, \chi) \mid \text{INT}(\phi)\end{aligned}$$

where c_i is a rational constant from $\mathbb{Q}$, x_i is a discrete variable, and v_i is a continuous variable. The functions NOT, OR, AND, and XOR are bit-wise logical operations, and they are only applicable to integers and assume a 2's complement format with arbitrary precision. The function INT converts a Boolean **true** value to an integer 1 and **false** value to an integer 0. Note that when continuous values are assigned to discrete variables, they are truncated (i.e., 13.5 becomes

[1] It should be noted that the traditional version of this example as a hybrid automata does not consider the software directly as this is cumbersome to do in that formalism.

[2] The rate of change is the first time derivative of the associated continuous variable.

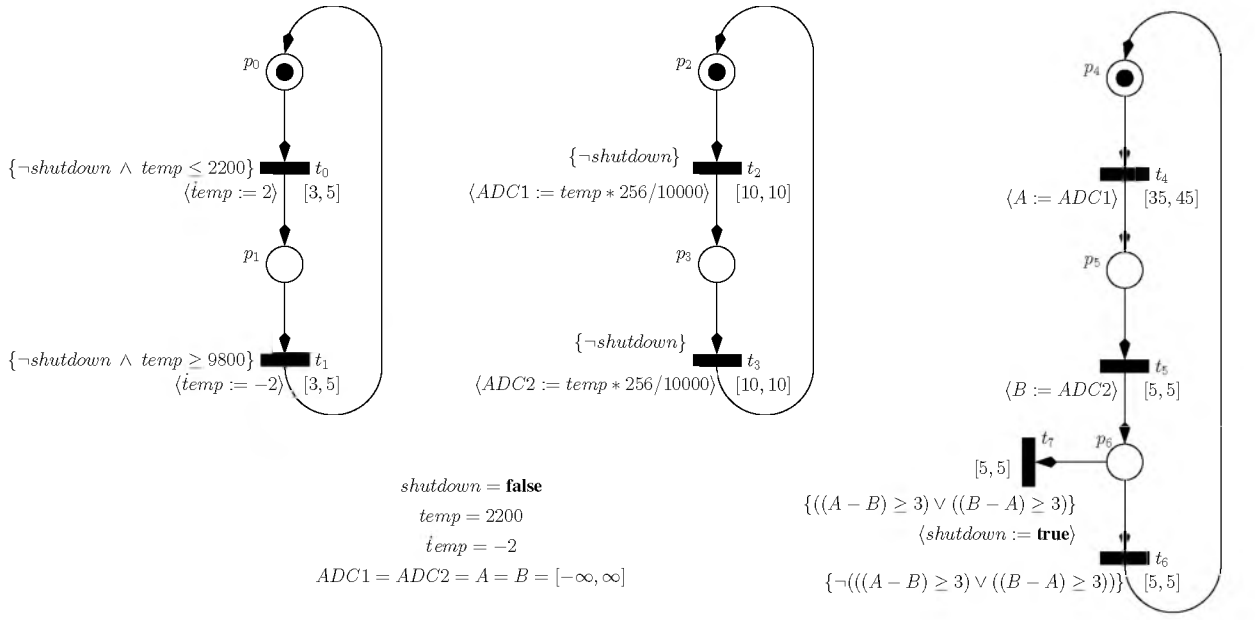

Fig. 2. Extended LHPN for the reactor example. The process on the left models the temperature of the reactor core (environment). The process in the center models the ADC subsystem of the microcontroller (hardware). The process on the right models the embedded software running on the microcontroller. Listed at the bottom left are the initial values of the system variables.

13). The set $\mathcal{P}_\chi$ is defined to be all formulas that can be constructed from the χ grammar.

The Boolean part of the grammar is as follows:

$$\begin{aligned}\phi \quad ::= \quad & \textbf{true} \mid \textbf{false} \mid b_i \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \chi = \chi \mid \\ & \chi \geq \chi \mid \chi > \chi \mid \chi \leq \chi \mid \chi < \chi \mid BIT(\chi, \chi)\end{aligned}$$

where b_i is a Boolean variable, and $BIT(\alpha_1, \alpha_2)$ extracts bit α_2 from α_1.[3] The set $\mathcal{P}_\phi$ is defined to be all formulas that can be constructed from the ϕ grammar.

The analysis algorithm restricts enabling conditions to a subset of the χ and ϕ grammars. The numerical part of this restricted grammar, χ_e, is defined as follows:

$$\begin{aligned}\chi_e \quad ::= \quad & c_i \mid x_i \mid (\chi_e) \mid -\chi_e \mid \chi_e + \chi_e \mid \chi_e - \chi_e \mid \\ & \chi_e * \chi_e \mid \chi_e / \chi_e \mid \chi_e \hat{} \chi_e \mid \chi_e \,\%\, \chi_e \mid \\ & \text{NOT}(\chi_e) \mid \text{OR}(\chi_e, \chi_e) \mid \text{AND}(\chi_e, \chi_e) \mid \\ & \text{XOR}(\chi_e, \chi_e)\end{aligned}$$

This grammar does not allow continuous variables to be used, nor does it allow Boolean expressions to be converted into integers. The set $\mathcal{P}_{\chi_e}$ is defined to be all formulas that can be constructed from the χ_e grammar. The Boolean part of this restricted grammar, ϕ_e, is defined as follows:

$$\begin{aligned}\phi_e \quad ::= \quad & \textbf{true} \mid \textbf{false} \mid b_i \mid \neg\phi_e \mid \phi_e \wedge \phi_e \mid \phi \vee \phi \mid \\ & BIT(\chi_e, \chi_e) \mid \chi_e = \chi_e \mid \chi_e \geq \chi_e \mid \chi_e > \chi_e \mid \\ & \chi_e \leq \chi_e \mid \chi_e < \chi_e \mid v_i \geq \chi_e \mid v_i \leq \chi_e\end{aligned}$$

The set $\mathcal{P}_{\phi_e}$ is defined to be all formulas that can be constructed from the ϕ_e grammar. Intuitively, enabling conditions only allow continuous variables to appear on the left side of relations of the form $v_i \geq \chi_e$ or $v_i \leq \chi_e$. This guarantees that the right side of these relations remains constant between transition firings as time advances.

Each transition in an LHPN is labeled with an enabling condition as well as a set of assignments. In particular, the labels permitted in LHPNs are represented using a tuple $L = \langle En, D, BA, XA, VA, RA \rangle$:

- $En : T \rightarrow \mathcal{P}_{\phi_e}$ labels each transition $t \in T$ with an enabling condition.
- $D : T \rightarrow \mathbb{Q} \times (\mathbb{Q} \cup \{\infty\})$ labels each transition $t \in T$ with a lower and upper delay bound, $[d_l(t), d_u(t)]$.
- $BA : T \times B \rightarrow \mathcal{P}_\phi$ labels each transition $t \in T$ and Boolean variable $b \in B$ with the Boolean assignment made to b when t fires.
- $XA : T \times X \rightarrow \mathcal{P}_\chi \times \mathcal{P}_\chi$ labels each transition $t \in T$ and discrete variable $x \in X$ with the discrete variable assignment, specified as a pair of expressions $[y_l(t, x), y_u(t, x)]$, that is made to x when t fires.
- $VA : T \times V \rightarrow \mathcal{P}_\chi \times \mathcal{P}_\chi$ labels each transition $t \in T$ and continuous variable $v \in V$ with the continuous variable assignment, specified as a pair of expressions $[a_l(t, v), a_u(t, v)]$, that is made to v when t fires.
- $RA : T \times V \rightarrow \mathcal{P}_\chi \times \mathcal{P}_\chi$ labels each transition $t \in T$ and continuous variable $v \in V$ with the continuous rate assignment, specified as a pair of expressions $[r_l(t, v), r_u(t, v)]$, that is made to v when t fires.

Note that vacuous assignments (i.e., assignments to the existing value) are not represented in the graphical representation.

Transition t_0 from the environmental process of Fig. 2 has an enabling condition of $\{\neg shutdown \wedge temp \leq 2200\}$.

[3]Only defined when the expressions α_1 and α_2 evaluate to integer values.

The delay of this transition varies from 3 to 5 time units. When t_0 fires, the rate, $temp$, is assigned to 2. The firing of t_2 results in a discrete variable assignment to $ADC1$ which sets its value to the value of the expression $temp * 256/10000$. Note that this assignment scales a continuous variable and assigns a truncated value to an integer. The firing of transition t_7 assigns the Boolean variable $shutdown$ to **true**. This example contains no assignments to continuous variables outside the initial state. The value of $temp$ changes continuously over time at its current specified rate of change.

IV. Semantics for Extended LHPNs

The state of an LHPN is defined using a 7-tuple of the form $\sigma = \langle M, S, Y, Q, R, I, C \rangle$ where:

- $M \subseteq P$ is the set of marked places;
- $S : B \rightarrow \{0, 1\}$ is the value of each Boolean variable;
- $Y : X \rightarrow \mathbb{Z}$ is the value of each discrete variable;
- $Q : V \rightarrow \mathbb{Q}$ is the value of each continuous variable;
- $R : V \rightarrow \mathbb{Q}$ is the rate of each continuous variable;
- $I : \mathcal{I} \rightarrow \{0, 1\}$ is the value of each continuous inequality.
- $C : T \rightarrow \mathbb{Q}$ is the value of each transition clock.

The set of continuous inequalities, $\mathcal{I}$, consists of all sub-expressions of the form $v_i \bowtie \alpha$ where $\bowtie$ is $\leq$ or $\geq$, and α is a member of the set $\mathcal{P}_{\chi_e}$. In this example, this includes $temp \leq 2200$ and $temp \geq 9800$. These inequalities are treated in a unique way because their truth values can change due to time advancement. Maintaining this set is not strictly necessary for the semantics, but it is convenient in several definitions and is used by the analysis method.

The current state of an LHPN can change either by the firing of an enabled transition or by time advancement. Every transition $t \in T$ has a *preset* denoted by $\bullet t = \{p \mid (p, t) \in F\}$ and a *postset* denoted by $t\bullet = \{p \mid (t, p) \in F\}$. A transition $t \in T$ is enabled when all of the places in its preset are marked (i.e., $\bullet t \subseteq M$), and the enabling condition on t evaluates to true (i.e., $Eval(En(t), \sigma)$ where the function $Eval$ evaluates an expression for a given state). The function $\mathcal{E}(\sigma)$ is defined to return the set of enabled transitions for the given state. When a transition t becomes enabled, its clock is initialized to zero. The transition t can then fire at any time after its clock satisfies its lower delay bound and must fire before it exceeds its upper delay bound (i.e., $d_l(t) \leq C(t) \leq d_u(t)$) as long as it remains continuously enabled. A transition is disabled any time one of the places in its preset becomes unmarked or its enabling condition evaluates to false. From a state σ, a new state σ' can be reached by firing a transition t found in $\mathcal{E}(\sigma)$. This new state is determined as follows:

- $M' = (M - \bullet t) \cup t\bullet$;
- $S'(b_i) = Eval(BA(t, b_i), \sigma)$
- $Y'(x_i) = Eval(y_l(t, x_i), \sigma)$
- $Q'(v_i) = Eval(a_l(t, v_i), \sigma)$
- $R'(v_i) = Eval(r_l(t, v_i), \sigma)$
- $I'(v_i \bowtie \alpha) = (Q'(v_i) \bowtie Eval(\alpha, \sigma))$
- $C'(t_i) = \begin{cases} 0 & \text{if } t_i \notin \mathcal{E}(\sigma) \wedge t_i \in \mathcal{E}(\sigma') \\ C(t_i) & \text{otherwise} \end{cases}$

In other words, the marking is updated, Boolean, discrete, continuous value, and continuous rate assignments associated with transition t are executed, the state of the continuous inequalities are updated, and the clocks associated with newly enabled transitions are reset to 0. Due to space limitations, to simplify the semantics, it is assumed that the lower and upper bound for all assignments are equal. Arbitrary ranges require some additional state to be maintained [15].

In a state σ, time can advance by any value τ which is less than $\tau_{\max}(\sigma)$. The value of $\tau_{\max}(\sigma)$ is the largest amount of time that may pass before a transition is forced to fire (i.e., the clock associated with it exceeds its upper bound) or an inequality changes value (i.e., for an inequality of the form $v_i \geq \alpha$, its continuous variable's value, v_i, crosses the value returned by its expression, α). This is defined as follows:

$$\tau_{\max}(\sigma) = min \begin{cases} C(t_i) - d_u(t_i) & \forall t_i \in \mathcal{E}(\sigma) \\ \frac{Eval(\alpha,\sigma) - Q(v_i)}{R(v_i)} & \substack{\forall (v_i \geq \alpha) \in \mathcal{I}. \\ I(v_i \geq \alpha) \neq (R(v_i) \geq 0)} \end{cases}$$

The new state, σ', after τ time units have advanced is defined as follows:

- $Q'(v_i) = Q(v_i) + \tau \cdot R(v_i)$
- $I'(v_i \bowtie \alpha) = \begin{cases} R(v_i) \bowtie 0 & \text{if } Q'(v_i) = Eval(\alpha, \sigma) \\ I(v_i \bowtie \alpha) & \text{otherwise} \end{cases}$
- $C'(t_i) = \begin{cases} 0 & \text{if } t_i \notin \mathcal{E}(\sigma) \wedge t_i \in \mathcal{E}(\sigma') \\ C(t_i) + \tau & \text{otherwise} \end{cases}$

All other parts of the state are unaffected.

Consider again the extended LHPN model for the reactor example shown in Fig. 2. The left process models the temperature of the reactor simply as a triangle wave which increases with a rate of two until it exceeds 9800 at which point it changes direction and decreases at a rate of two until it reaches 2200. The middle process models the ADC subsystem of the microcontroller which samples one of the temperature sensors every 10 time units with the results going into one of the internal 8-bit ADC registers. Note that perfect temperature sensors are assumed in that the same temperature value, $temp$, is sampled in both ADC inputs. The right process that models the software running on the microcontroller begins with a delay of 35 to 45 time units to represent instructions in the loop that are not related to checking the temperature sensors. Next, the software loads the value in $ADC1$ into register A followed 5 time units later with the value of $ADC2$ being loaded into register B. Finally, the difference is calculated between A and B, and if the absolute value of this difference is three or larger, the reactor is shut down. Otherwise, the software loop repeats.

To illustrate LHPN semantics, consider a few states for the example in Fig. 2. In the initial state, $p0$, $p2$, and $p4$ are marked; $shutdown$ is **false**; $ADC1$, $ADC2$, A, and B

are undefined (i.e., $[-\infty, \infty]$); $temp$ is 2200 and changing at a rate of -2. In this state, transitions t_0, t_2, and t_4 are all enabled. Note that t_0 is guarded by the Boolean expression $\{\neg shutdown \wedge temp \leq 2200\}$ which is satisfied in the initial state. In the initial state, $\tau_{\max}$ is 5, since t_0 must fire within 5 time units. Let us assume that 4 time units pass. In this new state, the value of $temp$ is now 2192, since it has decreased at a rate of 2 for 4 time units. The clocks for t_0, t_2, and t_4 now all have the value of 4. Transition t_0 can fire at any point after its clock reaches a value of 3, and it must fire before its clock exceeds a value of 5. Since its clock now has a value of 4, t_0 can potentially fire. Alternatively, the value of $\tau_{\max}$ is now 1, so time can be advanced by any real value less than or equal to 1. Let us assume that transition t_0 fires resulting in the rate of change of $temp$ to be changed to 2. Note that since $temp \geq 9800$ is not true, transition t_1 does not become enabled. In this new state, $\tau_{\max}$ has a value of 6, since transition t_2 must fire after 6 more time units have passed. After 6 time units have passed, the value of $temp$ is now 2204 while the values of the clocks for t_2 and t_4 reach 10. At this point, transition t_2 must fire resulting in $ADC1$ taking the value 56. This firing also enables transition t_3, so its clock is reset to 0. From here, new states can continue to be found by advancing time and firing transitions.

V. State Sets

State space exploration is required to analyze and verify properties of LHPNs. This exploration is complicated by the fact that LHPNs typically have an infinite number of states. Therefore, to perform state space exploration on LHPNs, this infinite number of states must be represented by a finite number of convex state equivalence classes called *state sets*. State sets for extended LHPNs are represented with the tuple $\psi = \langle M, S, Y, Q, R, I, Z \rangle$ where:

- $M \subseteq P$ is the set of marked places;
- $S : B \rightarrow \{0, 1, \perp\}$ is the value of each Boolean variable;
- $Y : X \rightarrow \mathbb{Z} \times \mathbb{Z}$ is a range of values for each discrete integer variable;
- $Q : V \rightarrow \mathbb{Q} \times \mathbb{Q}$ is a range of values for each inactive continuous variable;
- $R : V \rightarrow \mathbb{Q} \times \mathbb{Q}$ is the current rate of change for each continuous variable;[4]
- $I : \mathcal{I} \rightarrow \{0, 1, \perp\}$ is the value of each continuous inequality;
- $Z : (T \cup V \cup \{c_0\}) \times (T \cup V \cup \{c_0\}) \rightarrow \mathbb{Q}$ is a *difference bound matrix* (DBM) [5] composed of active transition clocks, active continuous variables, and c_0 (a reference clock that is always 0).

State sets and states differ in several ways. First, entries in S and I are extended to be able to take the value of *unknown* ($\perp$) to indicate uncertainty in their value. Second, discrete integer and inactive continuous variables (i.e., $R(v_i) = 0$) are extended to allow them to take a range of values. Finally, a DBM Z is used to represent the ranges of values for clocks and active continuous variables. It should be noted that despite the use of state sets, due to the use of discrete and continuous variables, the state space of an LHPN may still be infinite making verification undecidable.

The use of state sets requires that the expression evaluation function, Eval(α, ψ), as well as the enabled transition function $\mathcal{E}(\psi)$, be extended to operate on ranges of values and to return a range of values. For example, the relational operators on ranges are defined as follows:

$$
\begin{aligned}
([l_1, u_1] = [l_2, u_2]) &= \text{if } (l_1 = l_2 = u_1 = u_2) \text{ then } 1 \\
&\quad \text{elseif } ((l_1 > u_2) | (l_2 > u_1)) \text{ then } 0 \\
&\quad \text{else } \perp \\
([l_1, u_1] > [l_2, u_2]) &= \text{if } (l_1 > u_2) \text{ then } 1 \\
&\quad \text{elseif } (l_2 \geq u_1) \text{ then } 0 \\
&\quad \text{else } \perp \\
([l_1, u_1] \geq [l_2, u_2]) &= \text{if } (l_1 \geq u_2) \text{ then } 1 \\
&\quad \text{elseif } (l_2 > u_1) \text{ then } 0 \\
&\quad \text{else } \perp \\
([l_1, u_1] < [l_2, u_2]) &= \text{if } (u_1 < l_2) \text{ then } 1 \\
&\quad \text{elseif } (u_2 \leq l_1) \text{ then } 0 \\
&\quad \text{else } \perp \\
([l_1, u_1] \leq [l_2, u_2]) &= \text{if } (u_1 \leq l_2) \text{ then } 1 \\
&\quad \text{elseif } (u_2 < l_1) \text{ then } 0 \\
&\quad \text{else } \perp
\end{aligned}
$$

When applying relational operators to ranges, the result may be "$\perp$" since the relational operator must be applied to all values in the range. For example, the statement $[1, 2] = [1, 2]$ returns "$\perp$" because the comparison is between all pairs of values in the ranges, not between the two ranges themselves.

Arithmetic on ranges has been well studied [8]. Addition and subtraction is fairly straightforward as shown below:

$$
\begin{aligned}
[l_1, u_1] + [l_2, u_2] &= [l_1 + l_2, u_1 + u_2] \\
[l_1, u_1] - [l_2, u_2] &= [l_1 - u_2, u_1 - l_2]
\end{aligned}
$$

However, dealing with the sign of the operands makes multiplication and division somewhat more complicated:

$$
\begin{aligned}
[l_1, u_1] * [l_2, u_2] &= [min(l_1 * l_2, l_1 * u_2, u_1 * l_2, u_1 * u_2), \\
&\quad max(l_1 * l_2, l_1 * u_2, u_1 * l_2, u_1 * u_2)] \\
[l_1, u_1] / [l_2, u_2] &= \begin{cases} [l_1, u_1] * [1/l_2, 1/u_2] & \text{if } 0 \notin [l_2, u_2] \\ [-\infty, \infty] & \text{otherwise} \end{cases}
\end{aligned}
$$

Division by a range that includes 0 is quite involved [8], so for simplicity a conservative unbounded range is returned.

The modulo, bitwise, and bit extraction operations on ranges cannot be easily performed exactly and may result in non-continuous ranges. For example, the operation $[6, 9]\%8$ generates the results 0, 1, 6, and 7. These can be grouped into the ranges $[0, 1]$ and $[6, 7]$, but this would require splitting the zone. One method to do this is to use a multizone DBM approach, as described in [3] which we plan to investigate in the future. Currently, a more conservative approach is taken, choosing the larger region $[0, 7]$ which encapsulates

[4] Note that although the rate is defined to be a range, the method requires the rate to be a single value. This is not a problem as an LHPN with ranges of rates can be transformed into one with only single valued rates [12].

all possible values. As another example, the bitwise AND operator may clear bits, but never results in new bits being set. Therefore, it never results in an answer greater than the larger of the operands. The AND of two negative numbers though can result in an even smaller negative number but never smaller than $l_1 + l_2$. To address these problems, these operations are performed exactly when the operands are single values (i.e., $l_i = u_i$), and the conservative approximations shown below are used when any operand is a range:

$$\begin{aligned}
[l_1, u_1]\%[l_2, u_2] &= [min(0, max(-(max(|l_2|, |u_2|) - 1), l_1)), \\
&\quad max(0, min(max(|l_2|, |u_2|) - 1, u_1))] \\
NOT([l_1, u_1]) &= [-(u_1 + 1), -(l_1 + 1)] \\
AND([l_1, u_1], [l_2, u_2]) &= [min(l_1 + l_2, 0), max(u_1, u_2)] \\
OR([l_1, u_1], [l_2, u_2]) &= [min(l_1, l_2), max(u_1 + u_2, -1)] \\
XOR([l_1, u_1], [l_2, u_2]) &= [min(l_1 - u_2, l_2 - u_1, 0), \\
&\quad max(u_1 + u_2, -(l_1 + l_2), -1)] \\
BIT([l_1, u_1], [l_2, u_2]) &= \bot
\end{aligned}$$

Any time abstraction is used, it is possible to capture invalid behaviors. False negatives can thus be found. Any error trace derived from an abstracted system must be scrutinized carefully to determine its validity.

VI. State Space Exploration

This section describes a state exploration method which uses *zones* that are defined using DBMs to represent the continuous portion of the state space. In particular, this section extends the state space exploration method for LHPNs described in [10], [12] to utilize the extended expression syntax for enabling conditions and assignments.

The DBM based method shown in Fig. 3 uses a depth first search to find the reachable state space for an extended LHPN. Note that because the state space of an LHPN may not have a finite representation, this is a semi-algorithm as it may not terminate. First, this method constructs the initial state set for the extended LHPN and adds it to the set of reachable state sets, Ψ. The initial state set is $\langle M_0, S_0, Y_0, Q_0, R_0, I_0, Z_0 \rangle$ where I_0 contains the initial value for all continuous inequalities (i.e., $I_0(v_i \bowtie \alpha) = (Q_0(v_i) \bowtie Eval(\alpha, \psi_0))$), and Z_0 includes active continuous variables (i.e., $R_0(v_i) \neq 0$) set to their initial value and clocks for enabled transitions set to zero. Next, the method uses the `findPossibleEvents` function to determine all possible events, E, that can result in a new state set. A single event, e, is arbitrarily chosen from E using the `select` function. If after removing e from E, events still remain in E, the remaining events and the current state set are pushed onto the stack for later exploration. At this point, the current state set, ψ, is updated to reflect the occurrence of the event, e. If this new state set, ψ', has not been seen before, it is added to the state space, Ψ, a new set of possible events is calculated, and the exploration continues from this new state. If the state set is not new, a previously explored state set and set of unexplored events are popped from the stack, and the exploration continues from this point. Finally, when the stack is found to be empty, the entire reachable state space has been found, and it is returned. This section now explains each of these steps in more detail.

```
reach()
  ψ = initialStateSet()
  Ψ = {ψ}
  E = findPossibleEvents(ψ)
  while(true)
    e = select(E)
    if (E − {e} ≠ ∅) then push(E − {e}, ψ)
    ψ' = updateState(ψ, e)
    if ψ' ∉ Ψ then
      Ψ = Ψ ∪ {ψ'}
      ψ = ψ'
      E = findPossibleEvents(ψ)
    else
      if(stack not empty) then (E, ψ) = pop()
      else return Ψ
```

Fig. 3. Semi-algorithm to find the reachable states.

The `findPossibleEvents` function shown in Fig. 4 determines the set of all possible events from the current state. There are two event types: a transition can fire or an inequality can change value due to the advancement of time. A transition may fire at any time after its clock has reached the lower bound of the delay for that transition, and it must fire before its clock exceeds the upper bound of its delay. Transition clocks become active when they become enabled, and, as mentioned before, only clocks for enabled transitions are kept in Z. Therefore, any transition whose clock is in Z (denoted $t \in Z$) that can reach its lower bound (i.e., $\mathrm{ub}(Z, t) \geq d_l(t)$) may fire. Note that $\mathrm{ub}(Z, t)$ is defined to retrieve the upper bound for t's clock from Z. An inequality, $v_i \bowtie \alpha$, may change value when it is possible for time to advance to the point where the value of the continuous variable, v_i, crosses the value of the expression, α. This is determined by the `ineqCanChange` function by examining the current state set, ψ. The `ineqCanChange` function must be modified from the one described in [12] since the original version only allowed α to be a rational constant. The new version must evaluate α based on the current state. It is important to note that α must be *relatively constant*. Namely, the value of α must only change as a result of transition firings. It is this requirement that led to the restrictions described earlier on the forms of expressions that can be used in enabling conditions. For each possible event, the `addSetItem` function is used to determine if this event can actually be the next to occur. The event may actually not be able to occur before some event already found in E, and it would not be added in this case. Alternatively, the event may be possible to occur next, and it may in turn prevent some other events in E from being next. The details of this function are the same as the previous version of the algorithm, so the interested reader should see [12].

The `updateState` function shown in Fig. 5 determines the new state set that is reached after the occurrence of an event, e. First, this function calls the `restrict` function to modify Z to reflect that time must have advanced to the point necessary for the event to have occurred (i.e., the clock for the transition firing reaches its lower bound, or the continuous variable v_i reaches the value of its right hand expression α). This function also must be extended

```
findPossibleEvents(ψ)
  E = ∅
  for t ∈ Z
    if ub(Z,t) ≥ d_l(t) then
      E = addSetItem(E,t)
  for (v_i ⋈ α) ∈ I
    if ineqCanChange(ψ,v_i,α) then
      E = addSetItem(E,(v_i ⋈ α))
  return E
```

Fig. 4. Algorithm to find possible events.

to address the fact that inequalities can now be bounded by expressions. Next, the `recanonicalize` function uses Floyd's all-pairs shortest path algorithm to restore Z to a canonical form. When the event is an inequality changing value, the next step simply updates its value in I. When the event is a transition firing, however, the state update is more involved as shown in Fig. 6 which is described below. Next, the transitions are checked to see if they have become newly enabled or disabled. A clock for a transition t not in Z that is enabled must be added to Z while a clock for a transition t in Z that is not enabled must be removed from Z. Here again is another necessary modification in that determining if a transition is enabled requires the evaluation of the more complex expressions that are allowed in extended LHPNs. Finally, time is advanced using the algorithm shown Fig. 7, Z is recanonicalized, and the new state set is returned.

```
updateState(ψ,e)
  Z = restrict(ψ,e)
  Z = recanonicalize(Z)
  if e ∉ T then
    ψ = updateIneq(ψ,e)
  else
    ψ = fireTransition(ψ,e)
  for t ∈ T
    if t ∉ Z ∧ t ∈ E(ψ) then
      Z = addT(Z,t)
    else if t ∈ Z ∧ t ∉ E(ψ) then
      Z = rmT(Z,t)
  Z = advanceTime(ψ)
  Z = recanonicalize(Z)
  return ψ
```

Fig. 5. Algorithm to update the state.

The `fireTransition` function shown in Fig. 6 is called by the `updateState` function to fire a transition t in state set ψ. This function must first update the marking by removing the tokens from all places in $\bullet t$ and adding tokens to all places in $t\bullet$. Next, the transition t is removed from Z. Then, all assignments labeled on t are performed. This includes Boolean variable, discrete variable, continuous variable, and rate assignments. For extended LHPNs, these assignment functions are more involved. While in the basic LHPNs only constants are assigned, in extended LHPNs these assignments involve more complex expressions which must be evaluated on the current state. The assignments may have changed the values of some inequalities, so these must be updated next. The rate assignments may have activated or deactivated a continuous variable, so all continuous variables are checked and added or removed from Z as necessary. Finally, Z is warped using `dbmWarp` to properly account for any rate changes that may have occurred. The warping function described in [10], [12] is a technique that allows zones to be used even when continuous variables evolve at non-unity rates. The warping function does not need to be changed for extended LHPNs, so the interested reader is referred to [10], [12]. Once again, the warping of zones is an abstraction of the state space which can result in false negatives. It does not, however, ever produce false positives, and it has been shown to be a reasonable abstraction allowing for accurate verification of several interesting systems [12].

```
fireTransition(ψ,t)
  M' = (M − •t) ∪ t•
  Z' = rmT(Z,t)
  S' = doBoolAsgn(ψ)
  Y' = doIntAsgn(ψ)
  (Z',Q') = doVarAsgn(ψ)
  R' = doRateAsgn(ψ)
  I' = updateI(S',Y',Q',R',I,Z')
  for v ∈ V
    if v ∉ Z ∧ R'(v) ≠ 0 then
      (Z',Q') = addV(Z',Q',v)
    else if v ∈ Z ∧ R'(v) = 0 then
      (Z',Q') = rmV(Z',Q',v)
  (Z',R') = dbmWarp(Z',R,R')
  return ⟨M',S',Y',Q',R',I',Z'⟩
```

Fig. 6. Algorithm to fire a transition.

The `updateState` function calls the `advanceTime` function, shown in Fig. 7, to advance time in Z. The basic idea behind this function is that it allows time to advance as far as possible without missing an event. To ensure that a transition firing t is not missed, `advanceTime` sets the upper bound value for the clock associated with t to the upper delay bound for t. To ensure that a change in inequality value is not missed on a variable v, all inequalities involving variable v are checked by the function `checkIneq`, and the largest amount of time that can advance before one of these inequalities changes value is assigned to the upper bound value for v. Note that this function must be modified to evaluate the expressions now found in these inequalities.

```
advanceTime(ψ)
  for t ∈ Z
    ub(Z,t) = d_u(t)
  for v ∈ Z
    ub(Z,v) = checkIneq(ψ,v)
  return Z
```

Fig. 7. Algorithm for advancing time.

VII. Case Study

We have updated the `LEMA` verification tool to support extended LHPNs as described in this paper. This includes an editor to create extended LHPNs. We have applied this updated version of `LEMA` to the fault-tolerant temperature sensor with several variations in parameter values. The results are shown in Table I. For each case, the number of

TABLE I

VERIFICATION RESULTS FOR THE REACTOR EXAMPLE.

Parameters	State sets	Runtime (s)	Verifies
Original	61469	2050	Yes
t_4 delay $[5, 15]$	10	0.035	No
9-bit ADCs	50028	598	No
t_2, t_3 delay $[20, 20]$	45325	268	No
$temp$ rates $[-4, 4]$	23235	180	No
$temp$ rates $[-4, 4]$, 7-bit ADCs	32636	603	Yes

state sets found, runtime in seconds, and whether it verifies to be correct are reported. Recall that the property being verified is that the reactor never shuts down since the temperature sensors are assumed to be perfect in the LHPN model.

The original model parameters shown in Fig. 2 verify to be correct in 2050 seconds (about 34 minutes) after finding 61469 state sets. If the delay between iterations of the software loop is reduced (i.e. the delay of transition t_4 is reduced to $[5, 15]$), it no longer verifies to be correct after finding 10 state sets in 0.035 seconds. The reason for this failure is that A and B can be loaded and compared before the ADC subsystem has sampled any valid temperatures. If the ADCs use 9-bits (i.e., 256 is replaced with 512 in the expressions for $ADC1$ and $ADC2$), the example again fails. In this case, there is too much resolution, and the discrete values of the temperatures can differ by three or more. If the sampling rate is decreased (i.e., the delays for t_2 and t_3 are changed to $[20, 20]$), the example fails, since the temperature can change too much between two subsequent samples. If the rate of temperature change is increased to $[-4, 4]$, the example fails because the temperature can again change too much between samples. Changing multiple parameters can make the example start to verify correctly again. For example, if both the rate of temperature change is increased to $[-4, 4]$ and the resolution of the ADCs is reduced to 7-bits, the reactor does not shut down. These results indicate that the correctness of this fault-tolerant temperature sensor is quite sensitive to parameter choices.

VIII. CONCLUSION

This paper proposes a formal model for the verification of embedded systems. This model enables the modeling of complete systems, including environmental sensor inputs. In particular, the LHPN model is extended to include discrete variables and expressions to check and modify them in order to represent registers and memory values in embedded software. This paper also presents a method for reachability analysis of extended LHPNs that is used to perform formal verification. Finally, a case study is presented for a fault-tolerant temperature sensor that includes both a continuous environment signal as well as discrete register values. Preliminary results on this case study are promising.

Recently, we have extended the LEMA tool to compile extended LHPNs from embedded software which can then be integrated with existing methods for creating LHPNs for analog/mixed signal circuits. The fault-tolerant temperature sensor presented in this paper is an abstraction of the complete model that can be derived using this compiler. To derive such an abstract model, we are investigating automatic methods to reduce the LHPN to include only the elements necessary to verify the property of interest. Since such abstractions as well as those in this paper can potentially result in false negative results, we are also developing techniques to analyze the error trace ultimately resulting in a complete abstraction-refinement verification method.

IX. ACKNOWLEDGEMENTS

This research is supported by SRC contracts 2005-TJ-1357 and 2008-TJ-1851, and an SRC Graduate Fellowship.

REFERENCES

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3 – 34, 1995. Hybrid Systems.

[2] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1992.

[3] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[4] R. David and H. Alla. On hybrid Petri nets. *Discrete Event Dynamic Systems: Theory and Applications*, 11(1–2):9–40, January 2001.

[5] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Proc. Automatic Verification Methods for Finite-State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 1989.

[6] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[7] E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *EMSOFT '08: Proceedings of the 7th ACM international conference on Embedded software*, pages 255–264, New York, NY, USA, 2008. ACM.

[8] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48(5):1038–1068, 2001.

[9] M.S. Jaffe, N.G. Leveson, M.P.E. Heimdahl, and B.E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, 1991.

[10] S. Little, N. Seegmiller, D. Walter, C. Myers, and T. Yoneda. Verification of analog/mixed-signal circuits using labeled hybrid Petri nets. In *Proc. International Conference on Computer Aided Design (ICCAD)*, pages 275–282. IEEE Computer Society Press, 2006.

[11] S. Little, D. Walter, and C. Myers. Analog/mixed-signal circuit verification using models generated from simulation traces. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2007.

[12] S. R. Little. *Efficient Modeling and Verification of Analog/Mixed-Signal Circuits Using Labeled Hybrid Petri Nets*. PhD thesis, University of Utah, May 2008.

[13] D. Walter, S. Little, C. Myers, N. Seegmiller, and T. Yoneda. Verification of analog/mixed-signal circuits using symbolic methods. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(12):2223–2235, 2008.

[14] D. C. Walter. *Verification of analog and mixed-signal circuits using symbolic methods*. PhD thesis, University of Utah, May 2007.

[15] David Walter, Scott Little, Nicholas Seegmiller, Chris J. Myers, and Tomohiro Yoneda. Symbolic model checking of analog/mixed-signal circuits. In *Proc. of Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 316–323, 2007.